DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Primer Trabajo Práctico

14 de Abril de 2010 — Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Bianchi, Mariano	92/08	bianchi-mariano@hotmail.com
Brusco, Pablo	527/08	pablo.brusco@gmail.com
Di Pietro, Carlos Augusto Lyon	126/08	cdipietro@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

El primer problema de este trabajo práctico consistió en dar un algoritmo que sea capaz de decir, dada una secuencia de números, la mínima cantidad de números a eliminar de la misma para que ésta sea unimodal. Esto es, que la secuencia cumple que es creciente desde el primer elemento hasta cierta posición y desde dicha posición es decreciente hasta el final[1].

Además, se estableció como requerimiento que la complejidad del algoritmo en cuestión fuera estrictamente menor que $O(n^3)$, siendo n el tamaño de la secuencia.

Se pensaron varias formas de encarar y resolver este problema, finalmente se aplicó la técnica de *programación dinámica*. En las sucesivas secciones, daremos una explicación detallada de cómo se implementó el algoritmo, cuál es su complejidad temporal y cómo se comportó frente a distintos valores y tamaños de entrada.

1.2. Explicación

Desde un comienzo se pensó que el problema podía resolverse usando *programación dinámica*, sin embargo comprender de qué forma debía aplicarse *principio de optimalidad* no fue, en principio, nada trivial. Tras un tiempo de analizar el problema, se pudo vislumbrar que el mismo podía ser visto como una variación de otro problema similar: el de encontrar dentro de una secuencia la subsecuencia creciente/decreciente más larga.
Básicamente, el razonamiento empleado fue el siguiente:

Si para cada posición i de la secuencia se conoce:

- Longitud de la subsecuencia creciente más larga desde el principio hasta i (que contenga al elemento i-ésimo)
- Longitud de la subsecuencia decreciente más larga desde i hasta el final (que contenga al elemento i-ésimo)

entonces es posible decir cuál es el largo de una secuencia unimodal que tiene como "pivote" u objeto más grande al i-ésimo de la secuencia. Finalmente, sólo bastaría averiguar cuál es el "pivote" que maximice esa longitud para de ésta forma resolver el problema.
Cabe observar, que determinar la longitud de la subsecuencia unimodal más larga es equivalente a encontrar la cantidad mínima de elementos a eliminar para transformar la secuencia dada en una secuencia unimodal. Este dato será tenido en cuenta a la hora de implementar el algoritmo.

A continuación se desarrolla el funcionamiento del algoritmo mediante el cual se logra resolver el problema planteado.

Para construir el algoritmo de programación dinámica se definieron:

[1] Debe ser estrictamente creciente/decreciente

- un vector denominado *ascenso* donde se va guardando, en la i-ésima posicion, la longitud de la subsecuencia más larga desde el inicio de la secuencia original hasta la posición i. Esto es:
  $ascenso_i$ = longitud de la secuencia creciente más larga que termina con el número v_i
- la siguiente relación de recurrencia:
  - $ascenso_0 = 0$
  - $ascenso_i = max_{j<i}\{ascenso_j + 1\}$ (con $v_j < v_i$)
- y la solucion final: $max_{1 \leq i \leq n}\{ascenso_i\}$

Para entender el algoritmo principal, se presenta el pseudocódigo de la función *completar_ascensos*, la cual completa el vector *ascenso* con los valores correspondientes.

```
1 void completar_ascensos(ascenso: vector< nat >, v: vector< int >)
2 Complejidad: θ(tam(v)^2)
3 for (i = 0; i <= tam(v); i++);                    // θ(tam(v)^2))   [a]
4 do
5 |  ascenso[i] ← maximoAsc(ascenso,v,i) + 1;      // θ(i)
6 end
```

[a] esta complejidad se da ya que $\sum_{i=0}^{tam(v)} (i) = \frac{tam(v) * (tam(v) - 1)}{2} \in \theta(tam(v)^2)$

La función maximoAsc(ascenso,v,i) indica, conciderando el vector ascenso y el vector v, la maxima longitud de un ascenso hasta la posicion i-1 del vector tal que cumpla la siguiente propiedad:

Si j es el indice del ascenso donde se encuentra el resultado, tiene que ocurrir que: [2]

$$(\forall k \in [0...i), v[k] < v[i]) \rightarrow ascenso[j] > ascenso[k]$$

Por lo tanto, de esta manera podemos obtener para cada pivote, la longitud del mayor ascenso hasta esa posición.

Veamos un *ejemplo*:
$v = \{9, 5, 2, 8, 7, 3, 1, 6, 4\}$
Las subsecuencias ascendentes más largas son $\{2, 3, 4\}$ o $\{2, 3, 6\}$

sucesión	9	5	2	8	7	3	1	6	4
longitud	1	1	1	2	2	2	1	3	3

[2] ver pseudocódigo en el anexo

Aquí podemos observar que en cada posición de la fila longitud, se encuentra la longitud de la secuencia creciente más larga hasta ese punto.

Una vez realizado este proceso, podemos obtener, ejecutando un algoritmo equivalente (pero desde el final hacia el principio) que obtenga, para cada posición, la longitud de la secuencia decreciente más larga desde la posición hacia el final. Al ejecutar esté proceso sobre el ejemplo obtendríamos:

sucesión	9	5	2	8	7	3	1	6	4
ascendente	1	1	1	2	2	2	1	3	3
descendente	5	3	2	4	3	2	1	2	1

Luego, para decidir cual es el mejor pivot, nos basamos en dos filas extra implicitas que podemos ver a continuación :

sucesión	9	5	2	8	7	3	1	6	4
ascendente	1	1	1	2	2	2	1	3	3
descendente	5	3	2	4	3	2	1	2	1
suma	5	3	2	5	4	3	1	4	3
tam(S) - suma;	4	6	7	4	5	6	8	5	6

Donde $suma_i = ascendente_i + descendente_i - 1 \ \forall i \in [1..tam(s)]$
(esto quiere decir, la suma de las longitudes de la secuencia de mayor longitud creciente hasta i + la longitud de la secuencia de mayor longitud decreciente desde i). Se le resta 1 porque ambas secuencias incluyen al elemento de la posición i.

La fila tam(S)-suma representa, cuantos elementos NO fueron usados para esta "escalera".

Por lo tanto se busca el máximo valor de la fila suma, lo que es equivalente a buscar el mínimo en la fila tam(s)-suma que representa la cantidad minima de borrados a realizar para lograr formar la secuencia unimodal y ese es el resultado devuelto por nuestra función.

1.3. Análisis de la complejidad del algoritmo

A continuación se calculará la complejidad del algortimo implementado en la función *escalerar*, que es el que resuelve el problema que se planteo. Primero veamos el pseudocódigo de la función completa:

void **escalerar**(v: vector⟨int⟩)

```
1  Complejidad: θ(tam(v)^2)
2  var max: nat;
3  var ascenso: vector⟨nat⟩(tam(v));                                   // θ(tam(v))
4  var descenso: vector⟨nat⟩(tam(v));                                  // θ(tam(v))
5  completar_ascensos(ascenso,v);                                      // θ(tam(v)^2)
6  completar_descensos(descenso,v);                                    // θ(tam(v)^2)
7  for (i = 0; i ≤ tam(v); i++) ;                                      // θ(tam(v))
8  do
9  |  if (ascenso[i] + descenso[i] ≥ max) then max ← ascenso[i] + descenso[i];
   |  // θ(1)
10 end
11 retornar tam(v) − max + 1;                                          // θ(1)
```

Pseudocódigo de la función completar_descensos:

void **completar_descensos**(descenso: vector⟨nat⟩, v: vector⟨int⟩)

```
1  Complejidad: θ(tam(v)^2)
2  for (i = tam(v); i ≥ 0; i−−) ;                                      // θ(tam(v)^2)   [a]
3  do
4  |  descenso[i] ← maximoDes(descenso,v,i) + 1;                       // θ(tam(v) − i)
5  end
```

[a] esta complejidad se da ya que $\sum_{i=tam(v)}^{0} (tam(v) - i) = \sum_{i=0}^{tam(v)} (i) = \frac{tam(v) * (tam(v) - 1)}{2} \in \theta(tam(v)^2)$

Pseudocódigo de la función maximoAsc:

nat **maximoAsc**(ascenso: vector⟨nat⟩, v: vector⟨int⟩,i: nat)

```
1  Complejidad: θ(i)
2  var res: nat ;                                                      // θ(1)
3  res ← 0 ;                                                           // θ(1)
4  for (k = 0; k < i; k++);                                            // θ(i)
5  do
6  |  if (v[j] < v[i] and res < ascenso[j]) then res←ascenso[j];       // θ(1)
7  end
```

Aclaración: el pseudocódigo de la función competar_ascensos() se encuentra en la sección "Explicación" y la función maximoDes() es analoga a maximoAsc().

Como se ve en los pseudocódigos, sin depender de los valores ingresados en el vector v, la cantidad de operaciones es la misma para un mismo tamaño de entrada. Por lo tanto, podemos presumir que esta función esta acotada por arriba y por abajo por una función cuadratica a partir de un n_0 es decir, escalerar() $\in \Theta(tam(v)^2)$

En conclusión, creemos que nuestro algoritmo esta siempre acotado por arriba y por abajo por una función cuadratica en el tamaño de la entrada (tam(v)). Veremos al realizar las pruebas si esto se cumple en la practica.

1.4. Detalles de implementación

Para compilar el programa sólo hace falta ejecutar el comando make en consola desde la carpeta principal del ejercicio.

Modo de Uso El programa lee un archivo de entrada en la misma carpeta que el ejecutable con el nombre "Tp2Ej1.in" y genera un archivo llamado "salida.out" con la solucion.

1.5. Resultados

Para poder analizar el comportamiento del algoritmo, se desarrolló un generador de secuencias de tamaños determinados con valores enteros en un rango determinado.

Al realizar el análisis sobre la complejidad del algoritmo, notamos que sin importar los valores en la secuencia ni la cantidad de elementos a sacar para convertirla en unimodal, debería comportarse de manera identica para tamaños de secuencia similares.

Por lo tanto, veamos que entradas generamos:

1. 100 secuencias de tamaños desde 1 hasta 99 con elementos aleatorios entre -1000000 y 1000000.
2. 250 secuencias de tamaños desde 1 hasta 250 con elementos aleatorios entre -1000000 y 1000000.
3. 250 secuencias de tamaños desde 1 hasta 250 con elementos iguales a cero, de manera que el resultado sea eliminar todos menos un elemento para convertirla en unimodal.

Luego, se desprenden algunas hipótesis, las cuales enunciaremos a continuación:

- La cantidad de operaciones no debería ser afectada por la cantidad de elementos a borrar para convertir la secuencia.
- Para tamaños de entradas identicos, el algoritmo debería realizar exactamente la misma cantidad de operaciones.

A continuación veremos gráficos que muestran el comportamiento del algoritmo utilizado, en donde, utilizaremos las mismas funciones para acotar el algoritmo bajo el mismo método de medición. Es decir, si lo que medimos es cantidad de operaciones por ejemplo, utilizaremos las funciones

$$f(x) = 1500 + 1{,}03x^2 \quad (como\ cota\ superior)$$
$$g(x) = -2000 + 1{,}03x^2 \quad (como\ cota\ inferior)$$

Y si lo que medimos es tiempo utilizaremos

$$f(x) = 3{,}5x^2 \quad (como\ cota\ superior)$$
$$g(x) = -70000 + 3{,}5x^2 \quad (como\ cota\ inferior)$$

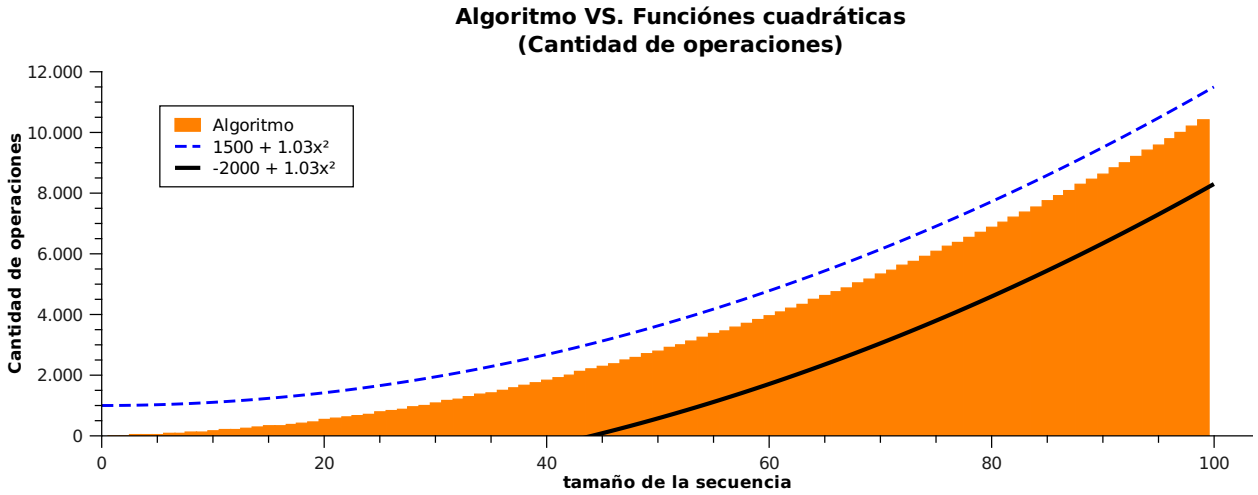


Gráfico 1 Cantidad de operaciones en secuencias de 1 hasta 100 elementos aleatorios comparados con funciones cuadráticas (se corresponde con la entrada generada número "1" de las enunciadas anteriormente).

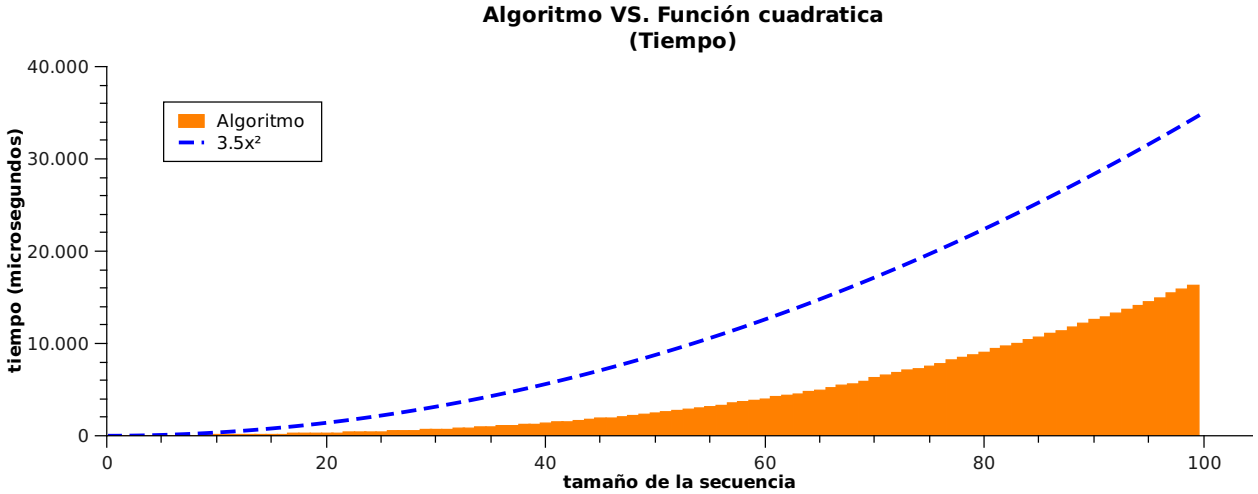


Gráfico 2: Tiempo de procesamiento en secuencias de 1 hasta 100 elementos aleatorios comparados con funciones cuadráticas (se corresponde con la entrada generada número "1" de las enunciadas anteriormente). La cota inferior propuesta no fue marcada ya que no era

significativa en este gráfico (ver el siguiente gráfico).

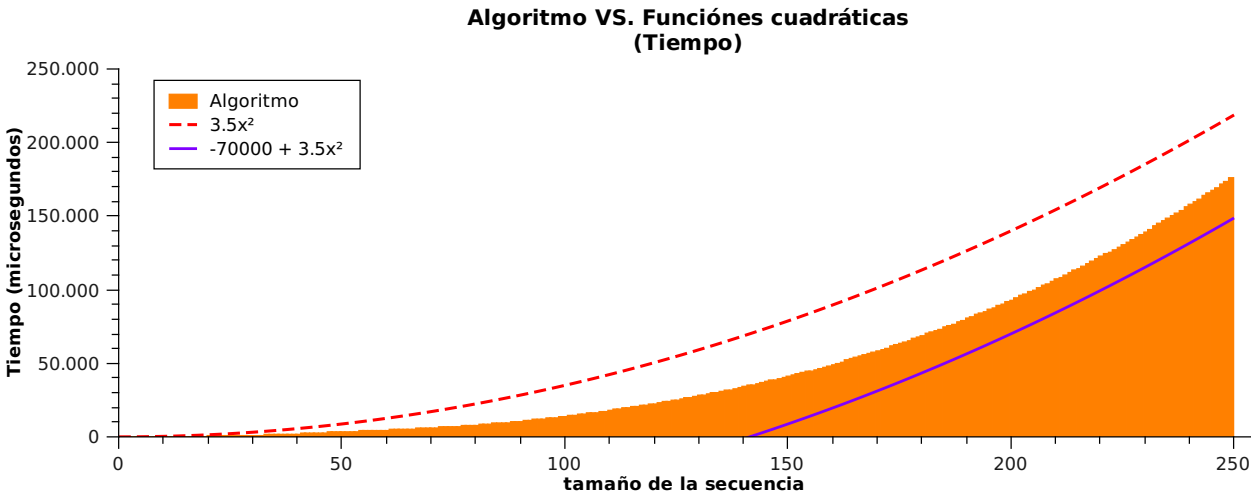


Gráfico 3: Tiempo de procesamiento en secuencias de 1 hasta 250 elementos iguales a cero comparados con funciones cuadráticas (se corresponde con la entrada generada número "3" de las enunciadas anteriormente).

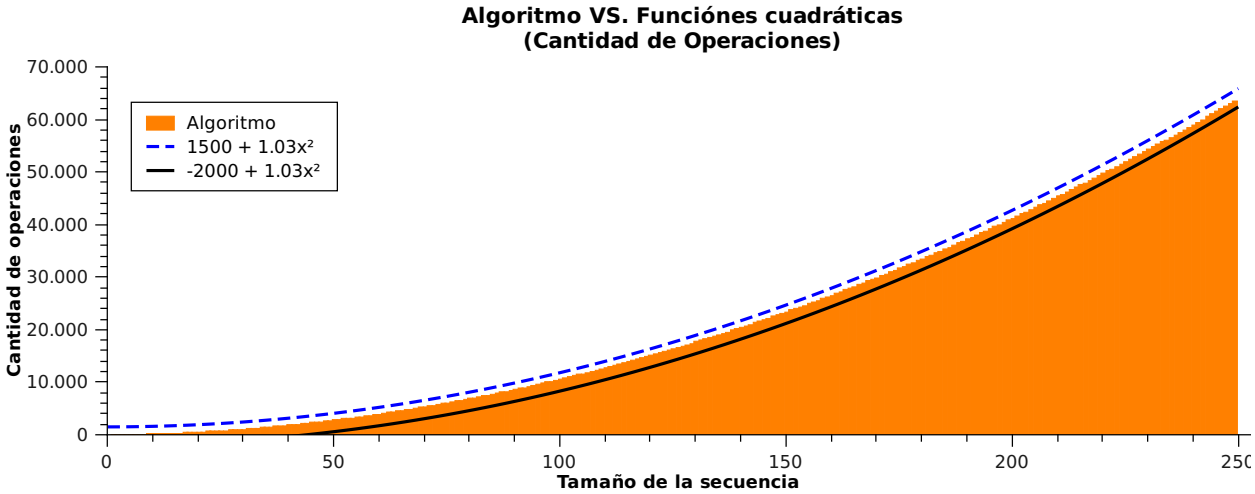


Gráfico 4: Cantidad de operaciones en secuencias de 1 hasta 250 elementos iguales a cero comparados con funciones cuadráticas (se corresponde con la entrada generada número "3"

de las enunciadas anteriormente).

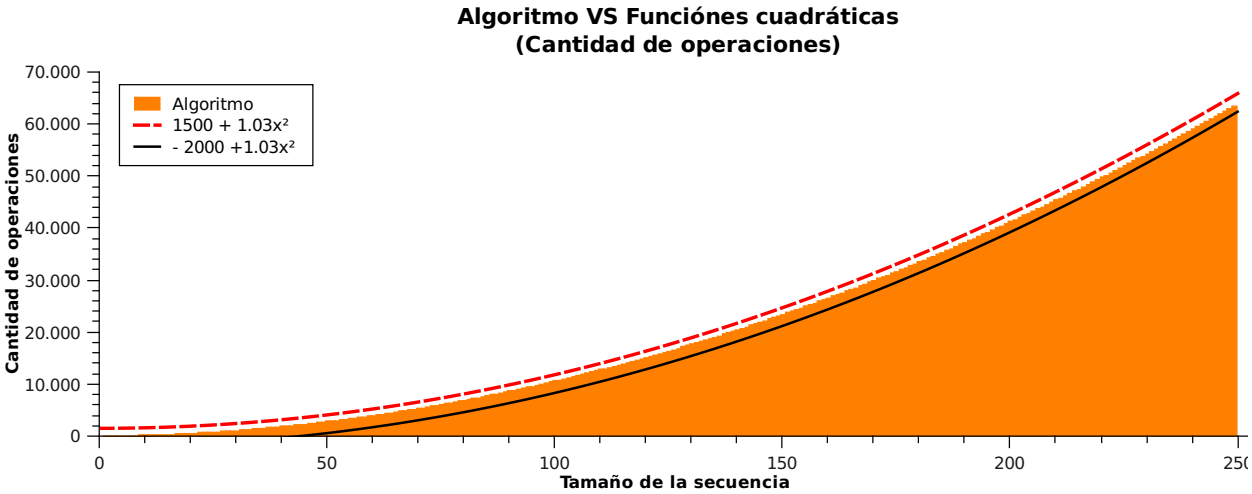


Gráfico 5: Cantidad de operaciones en secuencias de 1 hasta 250 elementos aleatorios comparados con funciones cuadráticas (se corresponde con la entrada generada número "2" de las enunciadas anteriormente).

Notar que para el mismo tipo de gráficos (de tiempo o de cantidad de operaciones), se utiliza las mismas funciónes cuadráticas para gráficar la cota superior e inferior que en el resto más alla de ser diferentes casos de pruebas.

1.6. Debate

Se puede apreciar en los gráficos propuestos en la sección [1.5] que en general la implementación realizada se comporta como se esperaba que en teoría lo hiciera. Es decir, durante el desarrollo del informe sobre este ejercicio, entre otras cosas, se postuló la hipótesis de que la complejidad del algoritmo tenía un costo de $\theta(tam(v)^2)$, donde $tam(v)$ es la cantidad de números en la secuencia.

Se puede identificar en cada gráfico una marcada similitud entre los resultados arrojados en las distintas mediciones que se hicieron sobre la implementación y parabolas (o funciónes cuadráticas) cuyas pendientes no varian para cada caso (medidos de la misma manera) que acotan por arriba y por abajo a la medición del algoritmo.

1.7. Conclusiones

Luego de describir el funcionamiento del algoritmo, de realizar las pruebas y gráficos pertinentes y de analizarlos detalladamente, podemos realizar algunas conclusiones.
Podemos asegurar fehacientemente que la complejidad del algoritmo propuesto es cuadratico y que la misma es $\theta(tam(v)^2)$ donde $tam(v)$ es la cantidad de elementos en la secuencia. Esto no sólo se desprende del análisis teórico realizado anteriormente, sino también de las sucesivas

pruebas realizadas. Claramente se puede observar en todos los gráficos presentados cómo el comportamiento de la implementación se asemeja a una función cuadrática sobre el tamaño de los datos de entrada.

Finalmente, si hacemos una comparación entre los resultados obtenidos al aplicarse el algoritmo sobre datos de entrada azarosos y datos de entrada que se condicen con lo que parecería ser un peor caso (todos ceros por ejemplo), se puede observar que la constante que acompaña a la complejidad para el peor caso es exactamente igual a la constante que aparece en el caso azaroso. Como una conclusión poco menos significativa entonces, se puede decir que todos los casos son mejor y peor caso para este algoritmo.

2. Ejercicio 2

2.1. Introducción

El segundo problema que se propone en este trabajo práctico es el de, dado el trazado de calles de una ciudad compueto por esquinas y calles, decidir si es posible orientar las calles de forma tal que dadas dos esquinas cualesquiera resulte posible llegar de la primera a la segunda.
Se pide además que la complejidad temporal para este problema, medida con modelo uniforme, sea estrictamente menor a $O(n^4)$.

Mediante un modelado con grafos, el problema puede ser representado de forma que los nodos de un grafo correspondan a las esquinas de la ciudad y las aristas a las calles que unen cada par de esquinas entre sí.
Entonces, para poder estudiar el problema con el modelo planteado es preciso primero establecer algunas definiciones:

- **Def$_1$:** Un grafo G es *fuertemente orientable* si existe una asignación de direcciones a los ejes del conjunto de ejes del grafo G tal que el digrafo resultante es fuertemente conexo.
- **Def$_2$:** *un puente o arista de corte* es una arista que al eliminarse de un grafo incrementa el número de componentes conexos de éste. Equivalentemente, una arista es un puente si y sólo si no está contenido en ningún ciclo.

A partir de las definiciones anteriores, se desprende que la solución al problema en cuestión debe establecer si el grafo que modela el trazado de las calles es fuertemente orientable. Esto es, decidir si es posible para cada instancia del problema orientar el grafo que la modela, para obtener así un digrafo fuertemente conexo de modo que sea posible llegar desde cualquier nodo hasta cualquier otro.

El algoritmo desarrollado para dar solución al problema planteado recorre los ejes del grafo evaluando si es posible o no orientarlo para que cumpla lo requerido. Para ello, el mismo, se basa los algoritmos tradicionales de recorrido de grafos, como son BFS y DFS. En la sección contigua se expone de forma detallada las ideas que dieron lugar al algoritmo así como también el algoritmo en sí.

2.2. Explicación

Tal como se expuso en la introducción de este informe, el algoritmo a implementar debía ser capaz de decidir si el grafo que modela el trazado de la ciudad es orientable de modo que sea factible conectar cualquier nodo con cualquiera de los nodos restantes; es decir, que el grafo pueda orientarse para ser un digrafo fuertemente conexo.

Si se observa la definición de digrafo fuertemente conexo, ésta nos dice que para cualquier par de nodos a y b, existe al menos un camino orientado desde a hasta b y al menos otro desde b hasta a. Esto a su vez, asegura que el grafo es conexo, ya que para cada nodo del grafo existe un camino orientado hacia cada uno de los nodos restantes.

A partir del análisis anterior, es esperable que el algoritmo devuelva que no es posible orientar el trazado de calles de la ciudad según lo pedido cuando se da alguna de las siguientes situaciones:

- existe al menos un par de nodos que no están unidos por un camino de calles, con lo cual el grafo que modela el problema no es conexo;
- se puede llegar de una esquina a a una esquina b, pero no es posible realizar el camino inverso, con lo cual el grafo que modela el problema tiene un puente.

Buscando formalizar esta idea se recurrió a distintas fuentes bibliográficas hasta que finalmente se hayo el siguiente teorema:

> Teorema: [3]
> Un grafo conexo G es fuertemente orientable si y solo si G no tiene puentes [4].

Del teorema anterior, se confirma la idea expresada en el analisis previo según la cual, si se verifica que el grafo es inconexo o que el grafo es conexo pero tiene un puente, entonces no es posible darle una orientación para convertirlo en un digrafo fuertemente orientado.
En línea con está pensamiento, el algoritmo que resuelve el problema fue desarrollado con la idea la de verificar que el grafo sea conexo y no tenga puentes. Para ello, el algoritmo va recorriendo el grafo al igual que lo hace el algoritmo DFS[5], con la salvedad de que cuando llega a un nodo nuevo, además de marcarlo como visitado, guarda en él la lista de ejes que conforman el camino hasta ese punto. Seguidamente, elije de entre los vecinos del nodo actual (excluyendo al padre), el próximo nodo hacia el cual va a dirigirse. En este paso pueden darse dos situaciones:

- que el nodo elegido no esté marcado
- que el nodo ya esté marcado, y no sea el padre del nodo actual

En el primer caso, como el nodo no está marcado, el algoritmo lo toma y realiza un paso recursivo sobre el mismo nodo para de este modo seguir recorriendo el grafo en busca de puentes o de inconexiónes . Por el contrario, en el segundo caso, al encontrarse con un nodo marcado pero que no es el padre del nodo actual, el algoritmo se encuentra frente a la presencia de un ciclo, ya que acaba de encontrar un camino que le permite llegar nuevamente hacia un nodo ya visitado. De ser éste el caso, procede a guardar todos los ejes del ciclo en un conjunto en el cual se van registrando todos los ejes que formen parte de algún ciclo. Así, el algoritmo procede hasta haber recorrido y marcado todos los nodos del grafo en cuyo caso finaliza devolviendo el número de nodos visitados y el conjunto con los ejes que pertenecen a algún ciclo.
Finalmente, si el numero de nodos recorridos es menor a la cantidad de nodos del grafo, eso significa que en algún punto el grafo no es conexo, y consecuentemente no es direccionable. De igual modo, si el conjunto de ejes que pertenecen a algún ciclo no es igual al conjunto de

[3] Demostrado por Robbins, 1939 - Gross & Yellen "Graph Theory", Teorema 11.1.4
[4] Véase la demotración del teorema en la sección Anexos
[5] Para conocer el funcionamiento del algoritmo DFS véase http://en.wikipedia.org/wiki/Depth-first_search

ejes del grafo, eso representa que existe al menos un eje que no forma parte de ningún ciclo, con lo cual es un puente y el grafo no puede ser dirigido según lo pedido.

A continuación se presenta el pseudocódigo del mismo, en el cuál usaremos a n como la cantidad de esquinas (o nodos) y a m como la cantidad de calles (o aristas):

void **dfs_ciclos**(vertice: $\mathbb{N}$, G: grafo&, cuenta: $\mathbb{N}$&)

```
1  Complejidad: O(m * n * log(n))
2  G.info[vertice].ejesHasta = G.info[G.info[vertice].padre].ejesHasta ;
   // O(tam(ejesHasta)) ⊆ O(n)
3  G.info[vertice].ejesHasta.insert(eje(G,vertice,G.info[vertice].padre)) ;   // O(1)
4  cuenta ← cuenta + 1 ;                                                      // O(1)
5  G.info[vertice].ejesHasta ← G.info[G.info[vertice].padre].ejesHasta ;      // O(n)
6  G.info[vertice].visitado ← true ;                                          // O(1)
7  var it*: it_secu <ℕ> ← G.info[vertice].vecinos.begin() ;                   // O(1)
8  for (it; it ≠ G.info[vertice].vecinos.end(); it++) do
9      if (G.info[*it].visitado == false);                                    // O(1)
10     then
11         G.info[*it].padre ← vertice ;                                      // O(1)
12         dfs_ciclos(*it,G,cuenta)
13     else
14         if (G.info[vertice].padre ≠ (*it));                                // O(1)
15         then
16         insertarResta(G.ejesUsados,G.info[vertice].ejesHasta
17         ,G.info[*it].ejesHasta);                                  // O(n * log(n))
18         G.ejesUsados.insert(eje(G,*it,vertice));  // O(log(tam(ejesUsados))) ⊆
           O(log(m)) ⊆ O(log(n^2)) ⊆ O(log(n))
19     end
20 end
```

Algorithm 1: Pseudocódigo de la función *dfs_ciclos* con el costo de cada instrucción en el modelo uniforme

void **insertarResta**(ejesEnCilos: set<int>, vEjesHasta: set<int>, wEjesHasta: set<int>)

```
1  Complejidad: O(n.log(n))
2  for (it = vEjesHasta.begin(), it ≠ vEjesHasta.end(), it++);
   // O(tam(vEjesHasta)) ⊆ O(n)
3  do
4      if (wEjesHasta.count(*it) == 0) then ejesEnCiclos.insert(*it);
       // O(log(tam(wEjesHasta))) ⊆ O(log(n))
5  end
```

Algorithm 2: Pseudocódigo de la función *insertarResta*

2.3. Análisis de la complejidad del algoritmo

Para el análisis de la complejidad de este algoritmo, vamos a remitirnos a los pseudocódigos de las funciones *dfs_ciclos* y *insertarResta* adjuntos en las sección [2.2]. En este caso se utilizará la misma notación para la cantidad de nodos y aristas que se utilizó en los pseudocódigos de la sección [2.2].

Si se observan las distintas complejidades que se dan dentro del algoritmo, se puede ver algunas que sobresalen sobre el resto. Es en estas en las que recae la complejidad total del algoritmo, por lo cuál el análisis se realizará sobre estas instrucciones.
Como primer caso de análisis, se puede observar en la tercer línea que la complejidad de la instrucción es $O(n)$. Esto se debe a que la asignación de conjuntos implementada en la *STL* tiene esta complejidad (destrucción y copia de un conjunto).

La siguiente complejidad a analizar es la referente al ciclo *for*. Como hay una llamada recursiva dentro, vamos a hacer un análisis más arraigado a la idea en sí de DFS que al pseudocódigo tal como está. Si pensamos en cómo funciona DFS, sabemos que este visitará a cada nodo a lo sumo una vez (estrictamente una única vez en caso que el grafo sea conexo). Por lo que si se observa detalladamente dentro del *for*, la sentencia *if* sólo se ejecutará[6] n veces. Dentro de esa sentencia, sólo se hacen operaciones de complejidad $O(1)$ y una llamada recursiva. Dicha llamada recursiva en realidad puede pensarse como $O(1)$ ya que si la función fuese iterativa, la recursión equivale a hacer un "apilar" de un nodo, y esto cuesta efectivamente $O(1)$.

Ahora falta ver cuántas veces se ejecutará la otra parte de la sentencia condicional, es decir, el *else*. La complejidad de ejecutar una vez esta parte de la sentencia está dada por el llamado a la función *insertarResta*, la cuál es $O(n*log(n))$. Si se analiza profundamente, se va a entrar a este lado de la sentencia siempre que se intente visitar a un nodo que ya fue visitado. En otras palabras y siendo un poco más formal, a cada nodo a lo sumo se va a intentar entrar tantas veces como ejes incidan en él, es decir, d(v) veces (donde v es algún nodo perteneciente al grafo). La primer vez que se lo visite entrará en el *if* y el resto de las d(v)-1 veces entrará en el *else*.
Sabemos que $\sum_{i=1}^{n} d(v_i) = 2*m$. Como el grado de un nodo puede ser a lo sumo n-1, entonces para cada nodo voy a estar ejecutando una vez el *if* y n-2 veces el *else*. Si contamos con que la complejidad de ejecutar el *if* es $O(1)$ y la del *else* es $O(n*log(n))$ y que por cada nodo se van a ejecutar una vez y n-2 veces cada una respectivamente (sin olvidarnos de la sumatoria antes mencionada), entonces se puede decir que en total, la complejidad de ejecutar el *if* es $O(n)$ y la del *else* es $O(2*m*n*log(n)) == O(m*n*log(n))$.

Finalmente entonces, la complejidad total del algoritmo es $O(n + m*n*log(n)) \subseteq O(m*n*log(n))$.

Hasta aquí hicimos el análisis suponiendo que la complejidad de *insertarResta* era $O(n*log(n))$. Pero para que la complejidad quede bien justificada, queda analizar que dicho algoritmo cumpla esta complejidad.

[6] Se entiende por "ejecutará" a que la condición de *true*.

La función consta de dos operaciones costosas que se repiten una cierta cantidad de veces. En primer instancia está el ciclo *for*, el cuál se va a ejecutar a lo sumo n-1 veces. Esto se debe a que dicho conjunto hace referencia a los ejes que pertenecen al camino simple raiz$\rightarrow v_i$ (con v_i cualquiera de los vértices pertenecientes al grafo original), por lo que la cantidad de ejes es a lo sumo n-1, es decir, $O(n)$. Luego, hay una sentencia condicional, para la cuál se hace una búsqueda sobre un conjunto, el cuál, al igual que el conjunto perteneciente al ciclo, tiene a lo sumo n-1 ejes por lo que dicha complejidad es $O(log(n))$. Finalmente, se realiza una inserción en otro conjunto. Este conjunto es el que va acumulando todos los ejes del grafo original que pertenecen a algún ciclo, por lo que a lo sumo puede contener todos los ejes del grafo, es decir m, por lo que la inserción cuesta $O(log(m))$. Pero m es a lo sumo n^2, es decir que: $O(log(m))$ $\subseteq O(log(n^2)) == O(2log(n)) == O(log(n))$. Es decir que por cada eje del primer conjunto, se ejecutan dos instrucciones de complejidad $O(log(n))$, por lo que la complejidad final de la función *insertarResta* sería $O(n * log(n))$.

2.4. Detalles de implementación

Dentro de la carpeta *./ej2/*, se puede encontrar el archivo ejecutable *ejercicio_2* compilado para GNU-Linux, el cual resuelve el problema anteriormente descripto. Este programa se ejecuta por consola mediante el comando `./ejercicio_2`, y recibe como parámetro los archivos de entrada *".in"* a procesar. Puede recibir tantos nombres de archivo como se desee, pero en caso de no recibir ninguno, el programa procesará el archivo *Tp2Ej2.in* que se encuentra incluído dentro de la misma carpeta.
Una vez ejecutado, el algoritmo procesa la cola de archivos que recibió como parámertos de a uno por vez generando para cada uno de ellos dos archivos:

- Un archivo *".out"* omónimo con la solucion a cada instancia del problema contenidos en el archivo de entrada.
- Un archivo omónimo con el sufijo *"_grafico.out"*, en el cual registra para cada instancia la cantidad de ejes del grafo que modela el problema y el tiempo promedio que tardó el algoritmo en resolverlo. Este archivo tiene objetivo facilitar la tarea de cargar los datos en el programa de análisis gráfico *QtiPlot*.

Ambos archivos son guardados en la misma carpeta de origen que la del archivo *'.in'*.

Por otra parte, en la misma carpeta, hay un Makefile el cual permite recompilar los archivos ejecutables con tan solo ejecutar el comando `make` en la consola. Además, ejecutando el comando `make clean` se pueden eliminar los archivos ejecutables y todos los archivos contenidos en la caperta *'test/'*.

Luego, en la carpeta *./ej2/sources* se encuentran el codigo fuente dell ejecutable antes descripto escrito en lenguaje C++ y cometando para simplificar la comprensión. Asímismo, en esta carpeta se puede hallar el script *input_gen2.py* para ser ejecutado desde la consola con el intérprete de Python mediante el comando `python input_gen2.py` . Al correr este programa se despliega un menú de opciones para generar distintos tipos de archivos *".in"* para ser resueltos por el ejecutable *ejercicio_2*. Una vez elegido el tipo de entrada a crear, el programa solicita que se ingrese la cantidad de casos a generar. Acto seguido guarda el

archivo generado en la carpeta *./ej2/tests*. Se recomienda, una vez generados varios archivos de prueba, ejecutar el comando `./ejercicio_2 tests/` para que el programa *ejercicio_2* resuelva todos los archivos de ese directorio con una única ejecución.

Finalmente, en *./ej2/* se hayan los archivos *.Tp2Ej2.in* y *Tp2Ej2.out* que vienen junto con en el enunciado de este Trabajo Práctico, y se haya también la carpeta *./ej2/pruebas_graficos* la cual contiene los archivos *".in"* generados para la exprimentacion que se presenta en este informe, junto con sus correspondientes archivos *".out"* y sus gráficos de m vs. *Tiempo*.

2.5. Resultados

El programa *input_gen2.py* fue desarrollado para poder crear cinco casos posibles de archivos input de distintas características, cada uno con un número de instancias dado por el usuario. Allí, cada instancia corresponde a un posible trazado de la ciudad caracterizado por la cantidad de esquinas y la cantidad de calles adyacentes a cada una de ellas.
A continuación se describen cuáles son esos casos y se explica brevemente que se esperaba observar en cada uno de ellos[7]:

a.- Casos con $0 \leq n \leq 100$ y $d(v_i) \geq \frac{n-1}{2}$, $\forall\ 0 \leq i \leq n$:
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente al que se presupone el peor caso posible (ya que en este caso están incluídos los grafos completos). Este es, el de instancias donde los grafos que las modelen tengan un gran número de nodos y sea sumamente densos (ya que cada nodo tiene al menos $\frac{n-1}{2}$ nodos incidentes).

b.- Casos con $0 \leq n \leq 100$ y $d(v_i) \leq \frac{n-1}{2}$, $\forall\ 0 \leq i \leq n$:
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente grafos con un gran número de nodos, pero no tan densos como el del caso previo (ya que cada nodo tiene a lo sumo $\frac{n-1}{2}$ nodos incidentes).

c.- Casos con $0 \leq n \leq 10$ y $d(v_i) \geq \frac{n-1}{2}$, $\forall\ 0 \leq i \leq n$:
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente a grafos con un reducido número de nodos, pero con muchas conexiones entre sí.

d.- Casos con $0 \leq n \leq 10$ y $d(v_i) \leq \frac{n-1}{2}$, $\forall\ 0 \leq i \leq n$:
Se pensó en este tipo de casos para poder evaluar el comportamiento del algoritmo frente al que se presupone el mejor caso posible. Este es el de grafos con un pequeño número de nodos y escazamente densos.

e.- Casos con $0 \leq n \leq 50$ y $d(v_i) \leq \frac{n-1}{2}$, ó $d(v_i) \geq \frac{n-1}{2}$, $\forall\ 0 \leq i \leq n$:
Finalmente, se pensó en este caso, para ver el comportamiento del algoritmo frente al caso de grafos con un número de nodos medio y con una densidad alta o baja establecida de manera aleatoria.

[7] Cabe aclarar que en todos los casos, tanto el número de nodos, como el grado de los mismos se generan de forma aleatoria, pero siempre dentro de los rangos establecidos por cada uno de los casos.

Mediante cada uno de estos casos se buscó estudiar el comportamiento del algoritmo para luego establecer su complejidad real, valiéndose para ello de la medición del tiempo (en microsegundos) que demora el algoritmo en resolver el cada instancia del problema. No obstante, puesto que este tipo de medición sufre de varias impresiciones propias del instrumento usado para medir se debió buscar una manera de acotar ese error[8].
Para subsanar este inconveniente, en cada uno de los archivos de pruebas utilizados durante la experimentación, se ejecutaron cien veces cada una de sus instancias, acumulando los tiempos de cada ejecución y calculando luego el tiempo promedio. De esta forma, se logró obtener un valor de estable y representativo del tiempo requerido por el algoritmo para resolver cada una de las intancias propuestas.

Seguidamente, y previo al momento de la experimentación, se formularon varias hipótesis en cuanto a cuál era el comportamiento esperable del algoritmo frente a cada uno de los tipos de inputs elegidos para su estudio. Las mismas son presentadas a continuación:

1) Tal como se explicó en la sección [2.3], la parte mas costosa del algoritmo está dada por la rama *else* del ciclo *for* del algoritmo *dfs_ciclos*. Esta rama es accedida cuando alguna arista del nodo sobre el que se está iterando incide sobre un nodo ya recorrido distinto del padre; es decir, cuando se encuentra una arista que es parte de algún ciclo. Luego si se desea estudiar el caso en que el algoritmo tiene la peor complejidad posible, es deseable que para el grafo a procesar cuente con muchos ciclos; más aún que el grafo sea completo. Por estos motivos, es de esperase que los archivos del caso `a` sean los que tarden más tiempo en resolverse, ya que son los de instancias con un gran número de nodos y una alta densidad.

2) Del mismo modo, y debido a su gran cantidad de nodos, resulta esperable que las instancias de los archivos del caso `b` tarde un tiempo considerable en resolverse, aunque menor que los del caso anterior.

3) Finalmente, se podría presuponer que debido a que tienen un n pequeño, en los grafos de las instancias de los archivos del caso `c` y `d` el grado de cada nodo también será un valor chico, disminuyendo así la posibilidad de formar ciclos y consecuentemente disminuyendo la posibilidad de entrar en la rama costosa del algoritmo. En conclusión, se puede pensar a priori que el algoritmo para estos casos resolverá las instancias a una velocidad aceptable/acelerada.

En lo que respecta a la experimentación propiamente dicha, se generaron cinco archivos correspondientes a cada uno de los tipos descriptos, cada uno con cien instancias (posibles trazados de la ciudad) del problema. Seguidamente, se procedió a correrlos con el programa *ejercicio_2*, obteniéndose para cada uno de ellos un archivo *"NombreDeArchivo.out"* y un archivo *"NombreDeArchivo_grafico.out"*; siendo este último el archivo en el cual se registraron de cada instancia del archivo original, la cantidad de aristas (m) y el tiempo promedio necesario para ser resuelta por el algoritmo.
Finalmente, haciendo uso de esos archivos se generaron, usando el programa de análisis gráfico

[8]Esto se debe a que mientras el ordenador está contando el tiempo de ejecución pueden tener lugar varios eventos como pueden ser las interrupciones al sistema operativo, llamados a memoria, etc., que detengan la ejecución del algoritmo en cuestión, pero no así la del contador de tiempo.

QtiPlot, diversos gráficos de m vs. *Tiempo* en los cuales se contrasta la curva de resultados con una función

$$f : \mathbb{N}_0 \to \mathbb{R} / f(n) \ = \ c * m * \sqrt{m} * log(m), \ c \in \mathbb{R}^+$$

para poder estudiar si la complejidad real se ajusta a la complejidad teórica.

A continuación se presentan los gráficos realizados para cada caso:

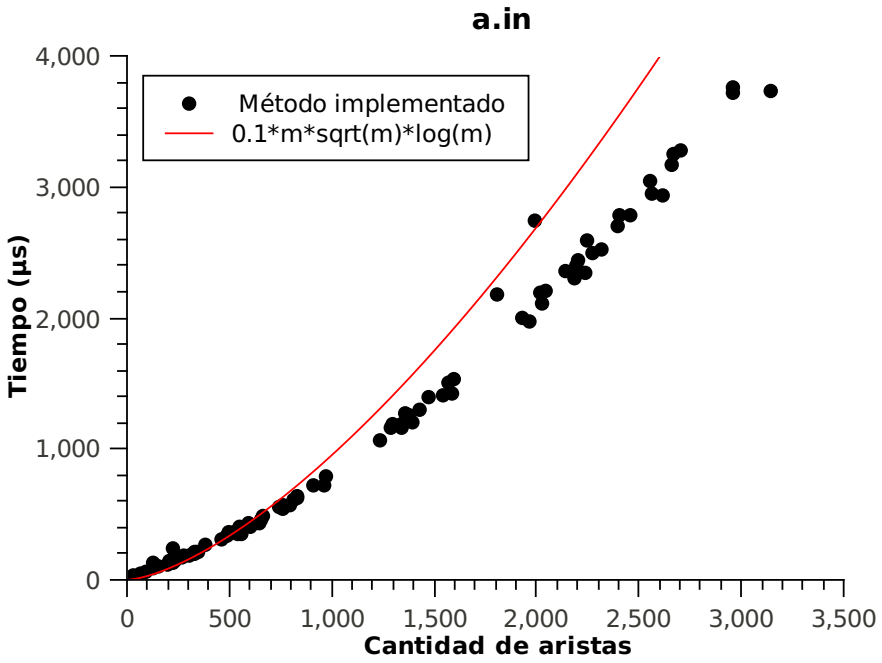


Gráfico A Gráfico de *Cantidad de ejes vs. Tiempo* del caso A

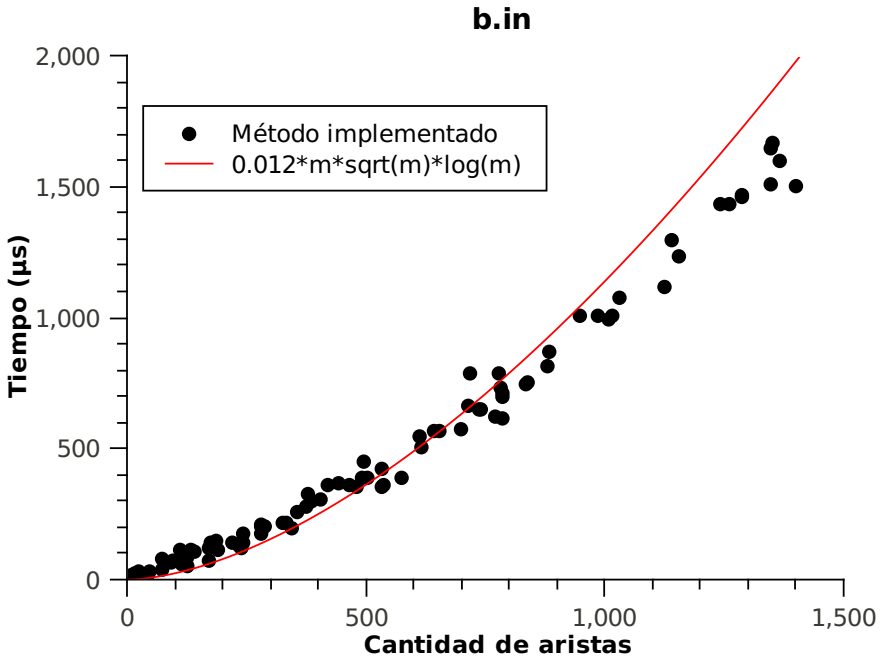


Gráfico B Gráfico de *Cantidad de ejes vs. Tiempo* del caso B

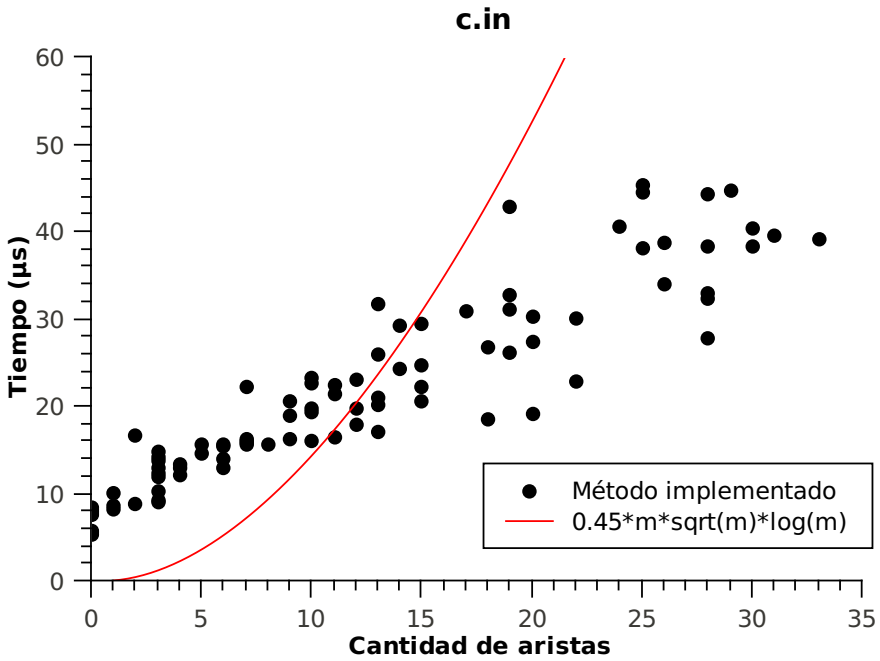


Gráfico C Gráfico de *Cantidad de ejes vs. Tiempo* del caso C

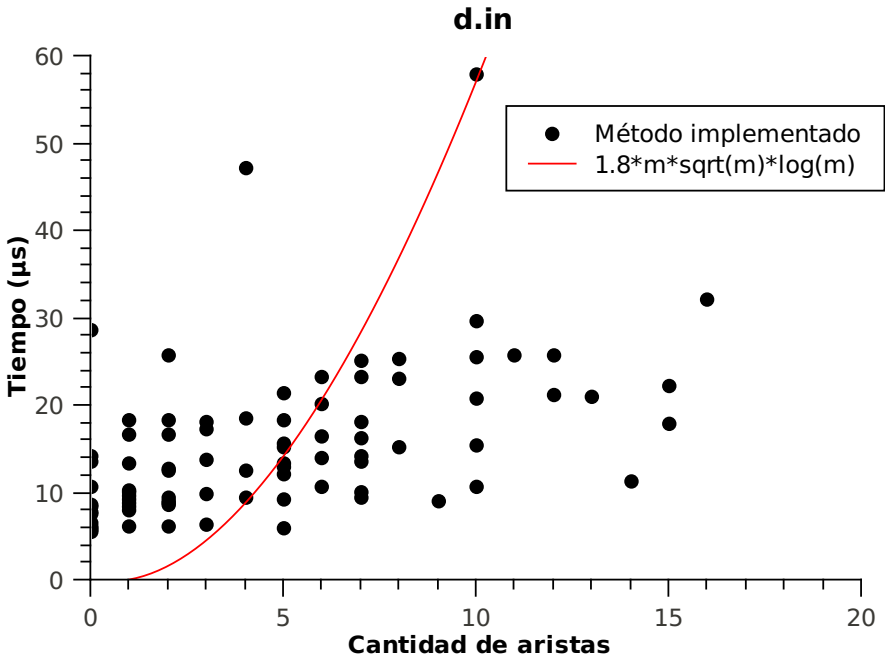


Gráfico D Gráfico de *Cantidad de ejes vs. Tiempo* del caso D

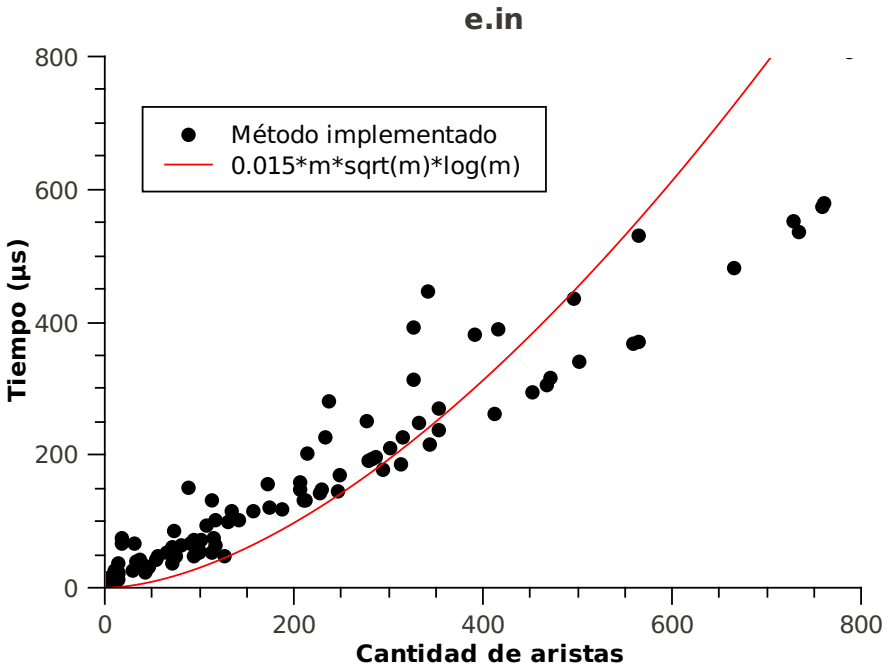


Gráfico E Gráfico de *Cantidad de ejes vs. Tiempo* del caso E

2.6. Debate

Como primera observación de los gráficos expuestos en la sección anterior, cabe decir que la complejidad real se ajustó a la complejidad teórica, ya que en todos los casos se pudo hallar una constante $c \in \mathbb{R}$ tal que a partir de algún n_0 el gráfico de $c * m * \sqrt{m} * log(m)$ pasa a acotar superiormente a la cantidad de operaciones realizadas por el algoritmo.

En lo que respecta a la las hipótesis postuladas en la sección [2.5], analicemos una a una su veracidad en función de los gráficos obtenidos.
Cotejando el gráfico A con los gráficos restantes se corrobora la hipótesis 1 ya que para las instancias del caso `a` los microsegundos consumidos por el algoritmo son del orden de los 3700μ s aproximadamente, mientras que el del caso más próximo en tiempo insumido (el caso `b`) no traspasa la barrera de los 1700μs.
De esta última observación, también se puede inferir que la hipótesis 2 se ve corroborada de manera empírica.

En cuanto a la hipótesis 3, la comparación de los gráficos C y D contra los de los gráficos A y B evidencia que los tiempos necesarios para resolver los archivos de los primeros son significativamente menor a los tiempos necesarios para resolver los archivos de los segundos. Esto en principio, parecería corroborar la hipótesis 3.
No obstante, se puede apreciar también que, si bien se puede encontrar una curva que acote estos gráficos a partir de un cierto punto, esta curva no se ajusta tan bien como las curvas que acotan los gráficos A y B. Asímismo, se puede ver que la curva de *m vs. Tiempo* de los gráficos C y D es mucho más irregular y oscilante que la de los gráficos A y B. Una comparación de los primeros con el gráfico E, arroja resultados similares, ya que si bien tiene valores de tiempo menor, este último gráfico se asemeja más a los gráficos de A y B.
En síntesis, si bien el resultado de la hipótesis 3 se cumple, las causas por las que esto ocurre no resultan del todo claras, y en principio no resulta correcto afirmar la hipótesis de que un número pequeño de nodos disminuye la posibilidad de formar ciclos y consecuentemente la posibilidad de entrar en la rama costosa del algoritmo.

2.7. Conclusiones

En base analizado en la sección [2.6], cabe remarcar como conclusión el hecho de que la complejidad medida en la realidad se asemeje bastante a las complejidades teóricas estimadas. Asímismo, el hecho de que las hipótesis 1 y 2 fueran corroboradas en base a los resultados arrojados por la experimentación.
Por último, se puede concluir que el hecho de que si bien se cumple el resultado de la hipótesis 3, las causas por lo que esto ocurre no se evidencian a partir de la experimentación. En consecuencia, no se puede afirmar la hipóteisis 3, pero tampoco se la puede descartar llanamente. Como hipótesis alternativa se puede plantear que la oscilación en las curvas de los gráficos C y D, o bien se debe al pequeño número de nodos del grafo (tal como se postuló en la hipótesis 3), o bien puede deberse al error producido al medir el tiempo aún pese haber tomado como medición representativa el tiempo promedio.

3. Ejercicio 3

3.1. Introducción

El tercer y último ejercicio de este trabajo práctico, consistía en dado un modelo de una prisión decidir si una persona condenada podía escapar o no de la misma. Este modelo fue dado de tal manera que se pudiera representar utilizando un grafo. La idea general del modelo de esta prisión, fue que en la misma haya habitaciones y pasillos. Cada pasillo conectaba dos habitaciones y no había más de un pasillo conectando dos habitaciones. Además, cada habitación podía estar vacía, tener una puerta o tener una llave. Para pasar por una habitación que tuviera una puerta había que tener su respectiva llave.

Como requerimiento para este problema, se especificó que su solución debía tener una complejidad estrictamente menor que $O(n^3)$, con n igual a la cantidad de habitaciones.

3.2. Explicación

En primer instancia, se pensó en alguna forma de recorrer el grafo de manera que la misma sirviera para solucionar el problema planteado. Inmediatamente, surgió la idea de utilizar el método BFS para dicho fin.

Seguidamente, se realizaron algunos ejemplos en papel, para ver de que forma se iba a comportar el método elegido para este modelo en cuestión, ya que se no podía utilizar un BFS normal por las restricciones con las que se contaba. Luego de observar este comportamiento, se encontró una forma de resolver el problema utilizando BFS, la cuál se explica a continuación.

La idea utilizada para este ejercicio resultó bastante simple. La misma consiste en los siguientes pasos:

- Se comienza a realizar BFS desde la primer habitación.
- Cada vez que se encuentra una habitación con una llave, ésta se guarda en un conjunto de llaves.
- Cada vez que se encuentra una habitación con puerta pueden suceder 2 cosas:
  - Si ya se había encontrado su llave, se la recorre como si no tuviese puerta.
  - Si no se había encontrado su llave, se la ingresa a una cola donde se alojaran las habitaciones con estas características, sin aplicar BFS en ella.
- Cuando el recorrido BFS termina, puede darse por dos motivos:
  - Que se hayan recorrido todas las habitaciones.
  - Que queden habitaciones sin recorrer en la cola de aquellas que tienen puerta y que cuando se las intentó visitar no se tenía su llave.

Una vez terminado el primer BFS, se verifica si ya se recorrieron todas las habitaciones o, caso contrario, si quedan habitaciones que se puedan visitar[9]. Si sucede lo segundo, entonces se repiten todos los pasos antes mencionados tomando como primer habitación para recorrer con BFS alguna de las que se habían encolado de la cuál se tenga su llave.
Si sucede que se recorrieron todas las habitaciones o que no hay ninguna llave disponible que abra las habitaciones encoladas, entonces se sale de la recursión y se verifica si se visitó o no la última habitación (o habitación de salida). Si se visitó entonces significa que hay forma de salir de la prisión y sino, significa lo contrario.

A continuación se adjunta, a modo de pseudocódigo, los algoritmos utilizados para resolver el problema descrito anteriormente. En los mismos usaremos a n como la cantidad de habitaciones (o nodos).

[9]esto es equivalente a ver si de todas las habitaciones que quedaron encoladas por tener una puerta a la cuál no se tenía acceso, hay alguna de las que sí se tenga llave ahora

bool resolver(carcel c)

```
1  Complejidad: O(n^2)
2  vector < bool > habitacionesYaVisitadas(c.cantHabitaciones);        // O(n)
3  queue < int > habitacionesLimites;
4  vector < bool > llavesEncontradas(c.cantHabitaciones);              // O(n)
5  int proximaHabitacion;
6  bool puedoSeguir = true;                                            // O(1)
7  int contador;
8  bool termine = false;                                               // O(1)
9  proximaHabitacion = 0;                                              // O(1)
10 while (puedoSeguir ∧ ¬termine) do
11 |  recorrerPorBFS(proximaHabitacion, habitacionesLimites,
   |  habitacionesYaVisitadas, llavesEncontradas, c);                  // O(n^2)
12 |  termine = habitacionesLimites.empty();                           // O(1)
13 |  if (¬termine);                                                    // O(1)
14 |  then
15 |  |  contador = habitacionesLimites.size();                        // O(1)
16 |  |  while (puedoSeguir &&
   |  |  (¬llavesEncontradas[habitacionesLimites.front()]));           // O(n)
17 |  |  do
18 |  |  |  habitacionesLimites.push(habitacionesLimites.front());     // O(1)
19 |  |  |  habitacionesLimites.pop();                                 // O(1)
20 |  |  |  contador−;                                                 // O(1)
21 |  |  |  if (contador == 0);                                        // O(1)
22 |  |  |  then
23 |  |  |  |  puedoSeguir = false;                                    // O(1)
24 |  |  |  end
25 |  |  end
26 |  |  proximaHabitacion = habitacionesLimites.front();             // O(1)
27 |  |  habitacionesLimites.pop();                                   // O(1)
28 |  end
29 end
30 if (habitacionesYaVisitadas[c.cantHabitaciones-1]);                // O(1)
31 then
32 |  return true
33 else
34 |  return false
35 end
```

void recorrerPorBFS(int proximaHabitacion, *queue* < *int* > & habitacionesLimites, vector< *bool* > & habitacionesYaVisitadas, vector< *bool* > & llavesEncontradas, carcel& c)

```
1  Complejidad: O(n^2)
2  queue < int > habitacionesProximas
3  habitacionesProximas.push(proximaHabitacion)
4  habitacionesYaVisitadas[proximaHabitacion] = true;                    // O(1)
5  int actual
6  while (!habitacionesProximas.empty());                                // O(n)
7  do
8  |  actual = habitacionesProximas.front();                            // O(1)
9  |  habitacionesProximas.pop();                                       // O(1)
10 |  if (c.tieneLlave(actual));                                        // O(1)
11 |  then
12 |  |  llavesEncontradas[c.dameLlave(actual)] = true;                 // O(1)
13 |  end
14 |  for (int i = 0; i<c.cantHabitaciones; i++);                       // O(n)
15 |  do
16 |  |  if (c.sonAdyacentes(actual,i) ∧ ¬habitacionesYaVisitadas[i]);
   |  |  // O(1)
17 |  |  then
18 |  |  |  if (c.tienePuerta(i) ∧ ¬llavesEncontradas[i]);              // O(1)
19 |  |  |  then
20 |  |  |  |  habitacionesLimites.push(i);                            // O(1)
21 |  |  |  else
22 |  |  |  |  habitacionesProximas.push(i);                           // O(1)
23 |  |  |  end
24 |  |  |  habitacionesYaVisitadas[i] = true;                         // O(1)
25 |  |  end
26 |  end
27 end
```

3.3. Análisis de la complejidad del algoritmo

Para el análisis de complejidad de los algoritmos que resuelven el problema dado, vamos a remitirnos a los pseudocódigos propuestos en la sección [3.2]. Como aclaración, cabe decir que en los siguientes párrafos vamos a referirnos a n como la cantidad de habitaciones en la prisión (cantidad de nodos del grafo).
En primer instancia vamos a analizar el pseudocódigo del algoritmo *recorrerPorBFS*. La complejidad del mismo está dada por un flujo *for* anidado dentro de otro flujo *while*. Es visible que el flujo *for* tiene una complejidad $\Theta(n)$ ya que todas las operaciones dentro suyo son de complejidad $O(1)$ y el mismo se ejecuta n veces. Quizás no sea tan claro por qué el flujo *while* tiene complejidad igual a $O(n)$. Esto se debe a la naturaleza del algoritmo BFS. En BFS se recorre cada nodo una única vez, y si vemos cuándo es que termina de ejecutarse dicho flujo, veremos que lo hace cuando se vacía la cola *proximasHabitaciones*. Esta cola puede

tener un largo de hasta n habitaciones (por lo dicho previamente sobre el método BFS) por lo que este flujo se ejecutará como máximo tantas veces como habitaciones haya.

Una vez realizado el análisis del algoritmo *recorrerPorBFS* sólo nos queda verificar qué complejidad cumple el algoritmo *resolver*. Si se observa con atención, puede notarse que la complejidad del mismo está dada por el llamado a la función *recorrerPorBFS*. Pero no sólo basta con decir esto. También hay un flujo *while* más adelante, el cuál solo tiene complejidad $O(n)$. Esto se debe a que dicho flujo se va a ejecutar a lo sumo tantas veces como elementos haya en la cola *habitacionesLimites*. Para saber cuántos elementos hay a lo sumo en esta cola debemos remitirnos a la función *recorrerPorBFS* y ver que aquí sólo se alojaran aquellas habitaciones en las cuales haya una puerta de la que no se tenga la llave correspondiente. Por esto podemos ver que claramente en esta cola habrá a lo sumo n habitaciones.
Sólo basta una aclaración para que la complejidad de este algoritmo quede completa y ésta hace referencia al flujo *while* que engloba a los flujos antes descritos. En un primer acercamiento, uno podría decir que este ciclo se ejecuta a lo sumo n veces, arrojando una complejidad de $O(n)$, haciendo que la complejidad total sea $O(n^3)$ si recordamos que este engloba a una llamada a *recorrerPorBFS* cuya complejidad había resultado ser $O(n^2)$. Pero en realidad, si se mira más detalladamente se puede observar que este flujo va a ejecutarse tantas veces como haga falta para que *recorrerPorBFS* pase por todos las habitaciones (o nodos), es decir, si por ejemplo en el primer llamado a la función *recorrerPorBFS* se pasa por todos los nodos, entonces el ciclo en cuestión va a ejecutarse una única vez. Este análisis entonces nos permite ver que en este algoritmo, la complejidad esta dada por la de la función antes mencionada. En conclusión, la complejidad del algoritmo es $O(n^2)$.

3.4. Detalles de implementación

Para compilar el programa sólo hace falta ejecutar el comando make en consola.

Modo de Uso En consola, utilizar el comando: `./3_ej [entrada] [salida]`.
Los parámetros "entrada" y "salida" son opcionales. Si no se los utiliza, el programa toma por default los archivos "Tp2Ej3.in" como entrada y "Tp2Ej3NUESTRO.out" como salida. En el caso que se los utilice, se le deben pasar 2 nombres de archivos de entrada y salida respectivamente. El archivo de entrada debe existir y debe tener un formato válido, es decir, debe tener el formato descrito en el enunciado de este trabajo práctico. El archivo de salida puede existir o no. En caso de existir, este será completamente sobreescrito.

3.5. Resultados

Para poder analizar el comportamiento del algoritmo propuesto como solución al problema dado fue necesario encontrar una forma de generar de manera automática grafos que lo modelen. Esto resulta de una gran dificultad y se torna mucho más complicado si se quieren generar grafos que lo modelen y que además cumplan con ciertas características. Por lo tanto, para generar los casos de prueba se recurrió a un método que utilice números aleatorios. Esto no nos permite elegir de manera flexible alguna característica particular para el modelo final, pero facilita mucho la construcción del generador automático.

Luego de realizar un análisis sobre el comportamiento de los algoritmos *resolver* y *recorrerPorBFS* notamos que en general siempre se van a obtener resultados similares. Para ello se realizaron distintas pruebas:

- Se generaron casos en los que la primer habitación estuviese conectada con la última. Esta se realiza debido a que ambos algoritmos dejan de iterar si, entre otras condiciones, ya se visitó la última habitación.
- Para otros casos, se puso la restricción que ninguna habitación podía estar conectada con la última. Esto significa que no había solución posible.
- Por último, se generaron casos completamente azarosos, con la única precaución que la primer y última habitación no estuviesen unidas entre sí (de otra forma, esto sería igual al primer item).

Cabe aclarar que para los casos en los que se analizó el tiempo que necesitaba la implementación para resolver el problema, se hicieron varias corridas por cada caso y luego se tomó un promedio del tiempo total. Esto se realizó para evitar en mayor medida algún outlier que pudiera aparecer por cualquier evento ajeno al algoritmo.

Finalmente, del análisis antes realizado se desprenden algunas hipótesis, las cuales enunciaremos a continuación:

1. Para los casos en los cuales no exista una comunicación directa entre la habitación 1 y la n-ésima, el algoritmo va a presentar una complejidad cuadrática.
2. En los casos en que exista un pasillo que comunique directamente la primer habitación con la última, el algoritmo debería arrojar una complejidad lineal.
3. En aquellos casos azarosos, la complejidad debería ser cuadrática pero en promedio será posible acotarla por una función lineal.

A continuación presentamos los gráficos provenientes de las pruebas realizadas:

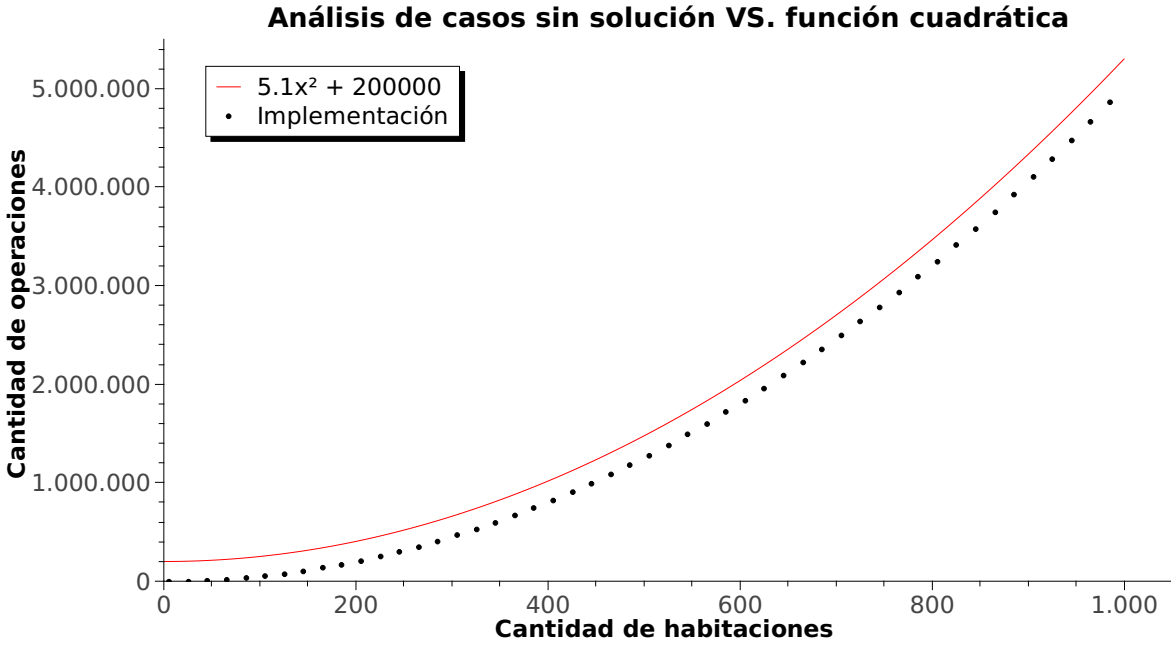


Figura 1: Muestra el comportamiento del algoritmo comparando cantidad de habitaciones contra cantidad de operaciones. Las entradas fueron creadas de modo que no exista un resultado posible, es decir, que ninguna habitación este conectada con la última.

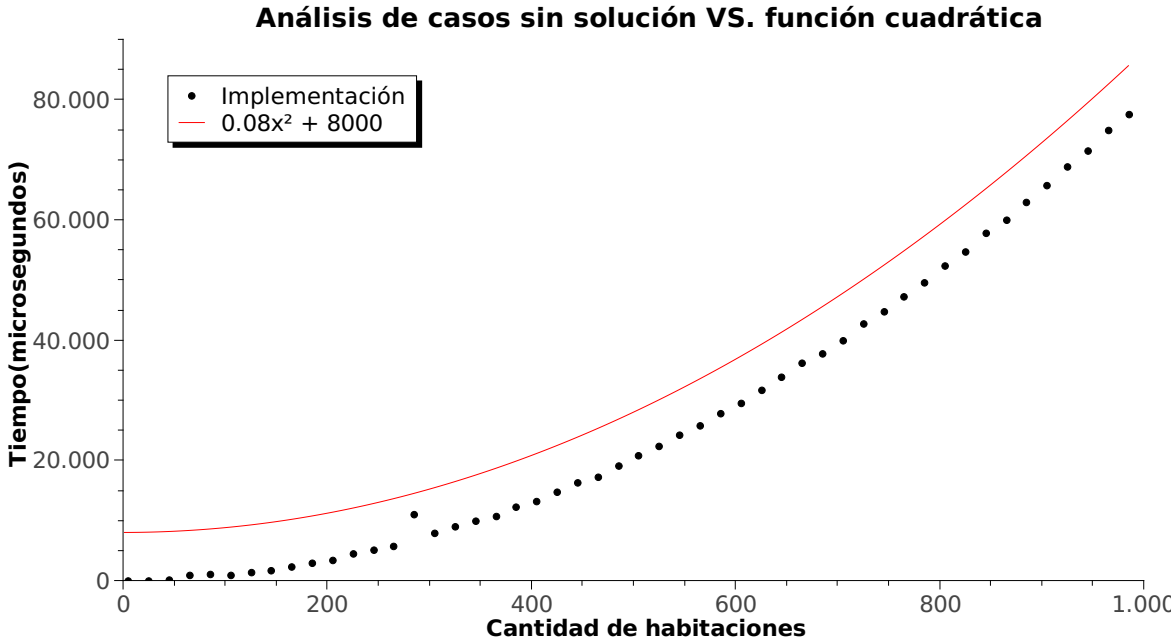


Figura 2: Muestra el comportamiento del algoritmo comparando cantidad de habitaciones contra tiempo de resolución. Las entradas fueron creadas del mismo modo que en la figura (1).

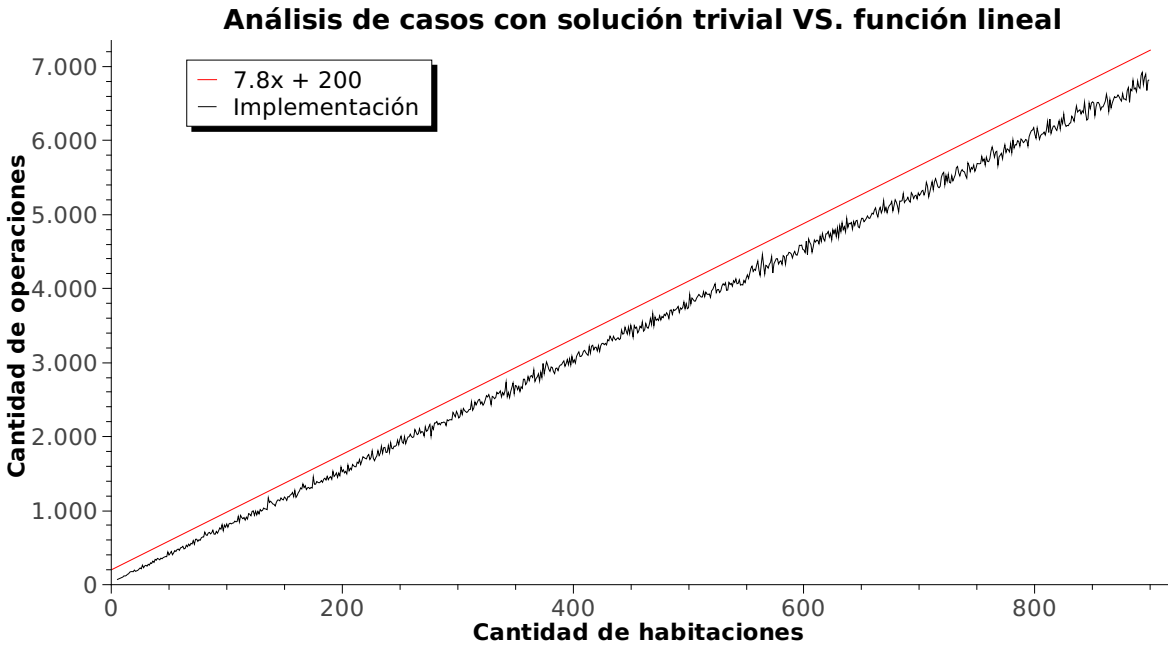


Figura 3: Muestra el comportamiento del algoritmo comparando cantidad de habitaciones contra cantidad de operaciones. Las entradas fueron creadas de modo que exista un resultado trivial, es decir, que la primer habitación este conectada con la última.

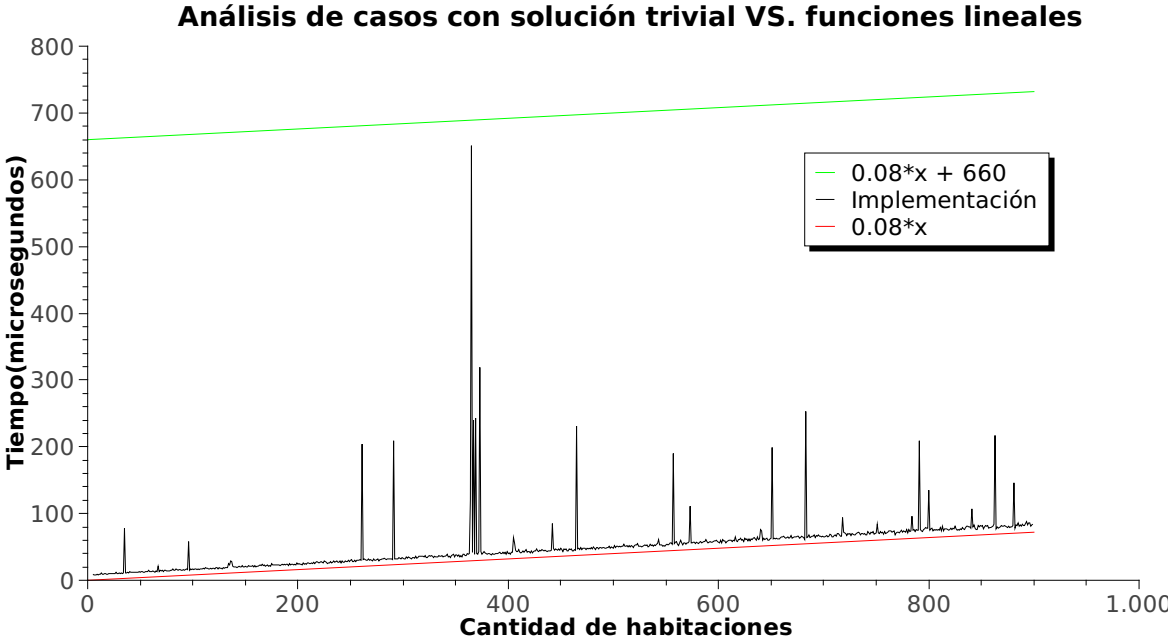


Figura 4: Muestra el comportamiento del algoritmo comparando cantidad de habitaciones contra cantidad de operaciones. Las entradas fueron creadas del mismo modo que en la figura (3).

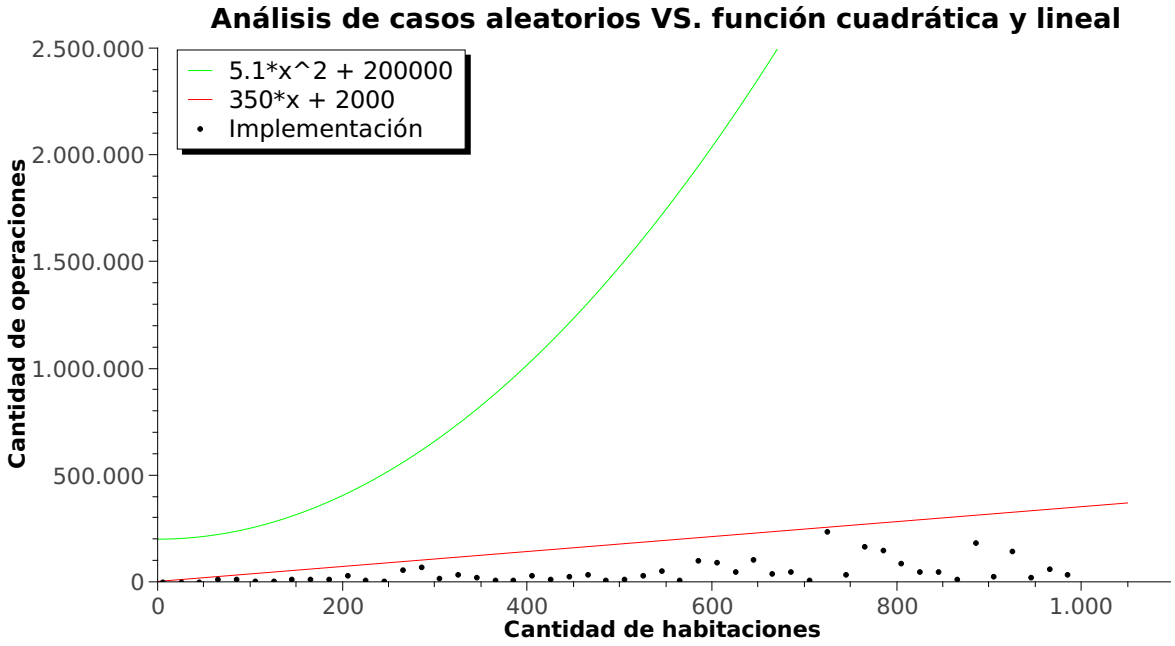


Figura 5: Muestra el comportamiento del algoritmo comparando cantidad de habitaciones contra cantidad de operaciones. Las entradas fueron creadas al azar pero de modo que no exista un resultado trivial, es decir, que la primer habitación no estuviera conectada con la última.

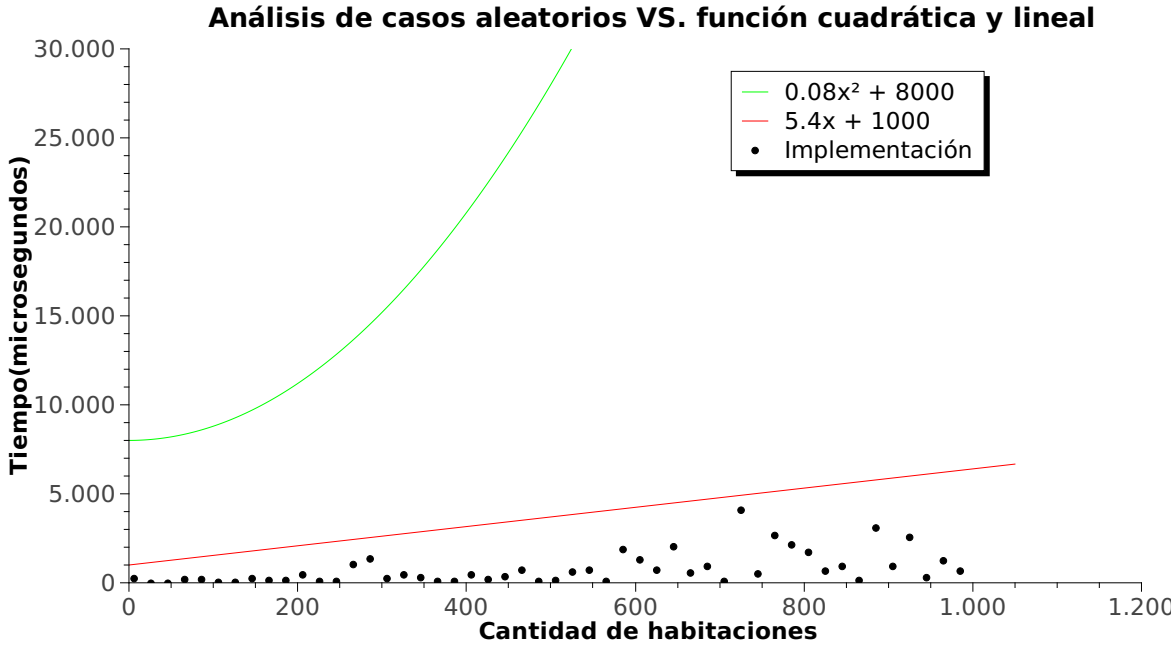


Figura 6: Muestra el comportamiento del algoritmo comparando cantidad de habitaciones contra cantidad de operaciones. Las entradas fueron creadas del mismo modo que en la figura (5).

3.6. Debate

De los resultados adjuntos en la sección anterior podemos observar algunos comportamientos particulares del algoritmo.
En los gráficos provenientes de los casos generados sin solución, se puede notar gran similitud entre los resultados arrojados durante la ejecución de la implementación y una función cuadrática. Esto puede deberse a que el algoritmo necesariamente va a recorrer todas las habitaciones posibles hasta concluir que no se visitó la última.

En el análisis de casos en los que hay una solución trivial, es decir en aquellos en los que la primer y última habitación son vecinas, los resultados tanto de tiempo como de cantidad de operaciones de la implementación parecieran estar acotados por una función lineal. Este comportamiento podría remitirse a cómo está implementado el algoritmo, en el cuál una vez que se visita a la última habitación no se continúan evaluando otras opciones sino que simplemente se devuelve un resultado positivo.

Por último tenemos aquellos casos en los que la disposición de las habitaciones y pasillos estaba totalmente realizada al azar salvo porque se "prohibía" la solución trivial. En estos gráficos se puede notar una disposición menos uniforme de los resultados provenientes de la implementación. Aquí se observa cómo estos pudieron ser acotados también por una función lineal.

3.7. Conclusiones

Si realizamos un análisis general de todo lo referente a este problema, podemos sacar algunas conclusiones. En general, en todos los casos de prueba que se analizaron, la implementación se comportó como se esperaba.
Se ve claramente cómo en el caso de las pruebas realizadas sobre entradas que no tenían solución, el algoritmo debe realizar muchas más operaciones que en los demás casos, ya que sólo deja de ejecutarse cuando se recorrieron todas las habitaciones alcanzables desde la primera. La cantidad de operaciones crece polinomialmente en comparación a la cantidad de habitaciones totales y por lo tanto también lo hace el tiempo que demora el algoritmo. Esto se debe a la forma en que se comporta el método BFS, cuya complejidad es $O(n^2)$, con n igual a la cantidad de nodos del grafo (en este caso, a la cantidad de habitaciones). Por esto, podemos decir que la primer hipótesis realizada fue correcta.

En el caso de la segunda hipótesis, pasa algo similar a lo anterior. Esta hace referencia al comportamiento del algoritmo cuando se utilizan entradas que contienen una solución trivial. En este contexto, es claro que por el modo de operar de la implementación, se encuentra una solución ni bien se analizan todas las habitaciones vecinas a la primera, ya que entre ellas se encuentra la última. Por lo tanto, es fácil ver aquí que el peor caso se da cuando la primer habitación está conectada con todas (este es el peor caso en este contexto, es decir, en el que existe solución trivial). Esto puede observarse además en los gráficos de la sección [3.5], donde los datos arrojados por la implementación son claramente acotados por una función lineal.

En última instancia, no podríamos decir que la tercer hipótesis fue corroborada, ya que para ello se debería realizar un análisis mucho más complejo y teórico sobre los datos de entrada, y esto excede los fines de este trabajo práctico. Sin embargo, sí se pudo comprobar que para todos los casos analizados los resultados fueron los esperados.

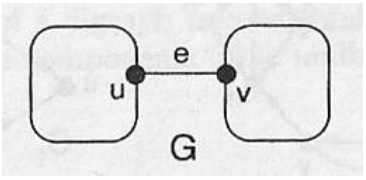
4. Anexos

4.1. Demostración del Teorema de Robbins

Teorema [Robbins, 1939]:
Un grafo conexo G es fuertemente orientable si y solo si G no tiene puentes.

Demostración: $\underline{\rightarrow)}$

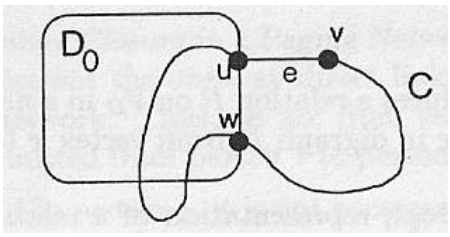
Utilizando el contrareciproco, supongamos que el grafo G tiene una arista de corte e que une a los vertices u y v. Entonces el unico camino entre u y v o v y u en el grafo G es e (ver figura). Por lo tanto para cualquier asignacion de direcciones, el nodo cola(e) nunca va a poder ser alcanzada por el nodo cabeza(e) (ver cuadro 1).



Cuadro 1:

Demostración: $\underline{\leftarrow)}$

Supongamos que G es un nodo conexo sin aristas de corte. Por esto toda arista en G cae en un circuito de G. Para hacer que G sea fuertemente orientable, empezaremos con cualquier circuito (D_0) de G y dirigiremos sus ejes en una dirección (obteniendo un circuito dirigido). Si el ciclo D_0 contiene todos los nodos de G, entonces la orientación esta completa (ya que D_0 es fuertemente conexo). De lo contrario, hay que elegir cualquier eje e uniendo a un vertice u en D_0 y a un vertice v en $V_G - V_{D_0}$ (ese eje existe ya que G es conexo). Sea C $= \langle u, e, v, ..., u \rangle$ un ciclo que contiene al eje e, y sea w el primer vertice luego de v en C que cae en el ciclo D_0 (ver cuadro 2).

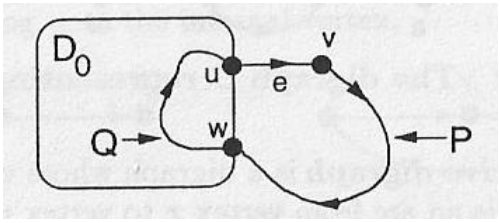


Cuadro 2:

A continuación, direccionamos los ejes de este camino entre v a w, obteniendo el camino dirigido $v-w$ que llamaremos P. Luego direccionamos el eje e desde u hasta v y consideramos el digrafo D_1 que se forma agregando el eje dirigido e a D_0 y todos los vertices y ejes dirigidos del camino P. Como D_0 es fuertemente conexo, entonces hay un camino dirigido de w a u (Q) en D_0 (ver cuadro 3). La concatenación de P con Q y el eje dirigido e forman un circuito

simple direccionado que contiene u y los nuevos vertices de D_1. (si los vertices u y w son el mismo, entonces P satiface ser un circuito simple dirigido)

Por lo tanto, el vertice u y todos estos nuevos vetices son mutuamente alcanzables en D_1 y además u y cada vectice del digrafo D_0 tambien son mutuamente alcanzables, y, por lo tanto , el digrafo D_1 es fuertemente conexo. Este proceso puede continuar hasta que el digrafo D_l para algun $l \geq 1$, contenga todos los vertices de G. En este punto, cualquier asignacion de direcciones hacia las aristas sin direccion restantes completaran la orientación de G, puesto que contendrá el digrafo fuertemente conexo D_l como subdigrafo.



Cuadro 3: